Lecture Notes: Program Analysis Correctness

17-355/17-665/17-819O: Program Analysis (Spring 2018)
Claire Le Goues and Jonathan Aldrich
clegoues@cs.cmu.edu, aldrich@cs.cmu.edu

The

1 Termination

As we think about the correctness of program analysis, let us first think more carefully about the situations under which program analysis will terminate. In a previous lecture, we analyzed the performance of Kildall's worklist algorithm. A critical part of that performance analysis was the the observation that running a flow function always either leaves the dataflow analysis information unchanged, or makes it more approximate—that is, it moves the current dataflow analysis results up in the lattice. The dataflow values at each program point describe an *ascending chain*:

Ascending Chain A sequence σ_k is an *ascending chain* iff $n \leqslant m$ implies $\sigma_n \sqsubseteq \sigma_m$

We can define the height of an ascending chain, and of a lattice, in order to bound the number of new analysis values we can compute at each program point:

Height of an Ascending Chain An ascending chain σ_k has finite height h if it contains $h+1$ distinct elements.

Height of a Lattice A lattice $(L, \sqsubseteq)$ has finite height h if there is an ascending chain in the lattice of height h, and no ascending chain in the lattice has height greater than h

We can now show that for a lattice of finite height, the worklist algorithm is guaranteed to terminate. We do so by showing that the dataflow analysis information at each program point follows an ascending chain. Consider the following version of the worklist algorithm:

```
forall (Instruction i ∈ program)
    σ[i] = ⊥
σ[beforeStart] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
    var thisInput = ⊥
    forall (Instruction j ∈ predecessors(i))
        thisInput = thisInput ⊔ σ[j]
```

```
let newOutput = flow(i, thisInput)
if (newOutput ≠ σ[i])
    σ[i] = newOutput
    worklist = worklist ∪ successors(i)
```

Question: what are the differences between this version and the previous version? Convince yourself that it still does the same thing.

We can make the termination argument inductively: At the beginning of the analysis, the analysis information at every program point (other than the start) is $\perp$ (by definition). Thus the first time we run each flow function for each instruction, the result will be at least as high in the lattice as what was there before (because nothing is lower in a lattice than $\perp$). We will run the flow function for a given instruction again at a program point only if the dataflow analysis information just *before* that instruction changes. Assume that the previous time we ran the flow function, we had input information σ_i and output information σ_o. Now we are running it again because the input dataflow analysis information has changed to some new σ_i'—and by the induction hypothesis, we can assume it is higher in the lattice than before, i.e. $\sigma_i \sqsubseteq \sigma_i'$.

What we need to show is that the output information σ_o' is at least as high in the lattice as the old output information σ_o—that is, we must show that $\sigma_o \sqsubseteq \sigma_o'$. This will be true if our flow functions are monotonic:

Monotonicity A function f is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

Now we can state the termination theorem:

Theorem 1 (Dataflow Analysis Termination)**.** *If a dataflow lattice $(L, \sqsubseteq)$ has finite height, and the corresponding flow functions are monotonic, the worklist algorithm will terminate.*

Proof. Follows the logic given above when motivating monotonicity. Monotonicity implies that the dataflow value at each program point i can only increase each time $\sigma[i]$ is assigned. This can happen a maximum of h times for each program point, where h is the height of the lattice. This bounds the number of elements added to the worklist to $h * e$, where e is the number of edges in the program's control flow graph. Since we remove one element of the worklist for each time through the loop, we will execute the loop at most $h * e$ times before the worklist is empty. Thus, the algorithm will terminate. □

2 Montonicity of Zero Analysis

We can formally show that zero analysis is monotone; this is relevant both to the proof of termination, above, and to correctness, next. We will only give a couple of the more interesting cases, and leave the rest as an exercise to the reader:

Case $f_Z[\![x := 0]\!](\sigma) = [x \mapsto Z]\sigma$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $[x \mapsto Z]\sigma_1 \sqsubseteq [x \mapsto Z]\sigma_2$

Case $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $\sigma_1(y) \sqsubseteq_{simple} \sigma_2(y)$
Therefore, using the pointwise definition of $\sqsubseteq$ again, we also obtain $[x \mapsto \sigma_1(y)]\sigma_1 \sqsubseteq [x \mapsto \sigma_2(y)]\sigma_2$

(α_{simple} and $\sqsubseteq_{simple}$ are simply the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.)

3 Correctness

What does it mean for an analysis of a WHILE3ADDR program to be correct? Intuitively, we would like the program analysis results to correctly describe every actual execution of the program. To establish correctness, we will make use of the precise definitions of WHILE3ADDR we gave in the form of operational semantics in the first couple of lectures. We start by formalizing a program execution as a trace:

Program Trace	A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, ...\}$ of program configurations, where $c_0 = E_0, 1$ is called the initial configuration, and for every $i \geqslant 0$ we have $P \vdash c_i \rightsquigarrow c_{i+1}$

.

Given this definition, we can formally define soundness:

Dataflow Analysis Soundness	The result $\{\sigma_i \mid i \in P\}$ of a program analysis running on program P is sound iff, for all traces T of P, for all i such that $0 \leqslant i < \mathit{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_{n_i}$

In this definition, just as c_i is the program configuration immediately before executing instruction n_i as the ith program step, σ_i is the dataflow analysis information immediately before instruction n_i.

Exercise 1. Consider the following (incorrect) flow function for zero analysis:

$$f_Z[\![x := y + z]\!](\sigma) = [x \mapsto Z]\sigma$$

Exercise 1. Give an example of a program and a concrete trace that illustrates that this flow function is unsound.

The key to designing a sound analysis is to make sure that the flow functions map abstract information before each instruction to abstract information after that instruction in a way that matches the instruction's concrete semantics. Another way of saying this is that the manipulation of the abstract state done by the analysis should reflect the manipulation of the concrete machine state done by the executing instruction. We can formalize this as a *local soundness* property:

Local Soundness	A flow function f is *locally sound* iff $P \vdash c_i \rightsquigarrow c_{i+1}$ and $\alpha(c_i) \sqsubseteq \sigma_i$ and $f[\![P[n_i]]\!](\sigma_i) = \sigma_{i+1}$ implies $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

In English: if we take any concrete execution of a program instruction, map the input machine state to the abstract domain using the abstraction function, find that the abstracted input state is described by the analysis input information, and apply the flow function, we should get a result that correctly accounts for what happens if we map the actual concrete output machine state to the abstract domain.

Exercise 2. Consider again the incorrect zero analysis flow function described above. Specify an input state c_i and use that input state to show that the flow function is not locally sound.

We can now show that the flow functions for zero analysis are locally sound. Although technically the overall abstraction function α accepts a complete program configuration (E, n), for zero analysis we can ignore the n component and so in the proof below we will simply focus on the environment E. We show the cases for a couple of interesting syntax forms; the rest are either trivial or analogous:

Case $f_Z[\![x := 0]\!](\sigma_i) = [x \mapsto Z]\sigma_i$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := 0]\!](\sigma_i) = [x \mapsto Z]\alpha(E)$
$c_{i+1} = [x \mapsto 0]E, n + 1$ by rule *step-const*
Now $\alpha([x \mapsto 0]E) = [x \mapsto Z]\alpha(E)$ by the definition of α.
Therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$, which finishes the case.

Case $f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\sigma_i$ where $m \neq 0$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\alpha(E)$
$c_{i+1} = [x \mapsto m]E, n + 1$ by rule *step-const*
Now $\alpha([x \mapsto m]E) = [x \mapsto N]\alpha(E)$ by the definition of α and the assumption that $m \neq 0$.
Therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$ which finishes the case.

Case $f_Z[\![x := y \ op \ z]\!](\sigma_i) = [x \mapsto ?]\sigma_i$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := y \ op \ z]\!](\sigma_i) = [x \mapsto ?]\alpha(E)$
$c_{i+1} = [x \mapsto k]E, n + 1$ for some k by rule *step-const*
Now $\alpha([x \mapsto k]E) \sqsubseteq [x \mapsto ?]\alpha(E)$ because the map is equal for all keys except x, and for x we have $\alpha_{simple}(k) \sqsubseteq_{simple} ?$ for all k, where α_{simple} and $\sqsubseteq_{simple}$ are the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.
Therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$ which finishes the case.

Exercise 3. Prove the case for $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$.

Now we can show that local soundness can be used to prove the global soundness of a dataflow analysis. To do so, let us formally define the state of the dataflow analysis at a fixed point:

Fixed Point	A dataflow analysis result $\{\sigma_i \mid i \in P\}$ is a fixed point iff $\sigma_0 \sqsubseteq \sigma_1$ where σ_0 is the initial analysis information and σ_1 is the dataflow result before the first instruction, and for each instruction i we have $\sigma_i = \bigsqcup_{j \in \texttt{preds}(i)} f[\![P[j]]\!](\sigma_j)$.

And now the main result we will use to prove program analyses correct:

Theorem 2 (Local Soundness implies Global Soundness)**.** *If a dataflow analysis's flow function f is monotonic and locally sound, and for all traces T we have $\alpha(c_0) \sqsubseteq \sigma_0$ where σ_0 is the initial analysis information, then any fixed point $\{\sigma_i \mid i \in P\}$ of the analysis is sound.*

Proof. Consider an arbitrary program trace T. The proof is by induction on the program configurations $\{c_i\}$ in the trace.

Case c_0:

$\alpha(c_0) \sqsubseteq \sigma_0$ by assumption.
$\sigma_0 \sqsubseteq \sigma_{n_0}$ by the definition of a fixed point.
$\alpha(c_0) \sqsubseteq \sigma_{n_0}$ by the transitivity of $\sqsubseteq$.

Case c_{i+1}:

$\alpha(c_i) \sqsubseteq \sigma_{n_i}$ by the induction hypothesis.
$P \vdash c_i \rightsquigarrow c_{i+1}$ by the definition of a trace.
$\alpha(c_{i+1}) \sqsubseteq f[\![P[n_i]]\!](\alpha(c_i))$ by local soundness.
$f[\![P[n_i]]\!](\alpha(c_i)) \sqsubseteq f[\![P[n_i]]\!](\sigma_{n_i})$ by monotonicity of f.
$\sigma_{n_{i+1}} = f[\![P[n_i]]\!](\sigma_{n_i}) \sqcup ...$ by the definition of fixed point.
$f[\![P[n_i]]\!](\sigma_{n_i}) \sqsubseteq \sigma_{n_{i+1}}$ by the properties of $\sqcup$.
$\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ by the transitivity of $\sqsubseteq$.

□

Since we previously proved that Zero Analysis is locally sound and that its flow functions are monotonic, we can use this theorem to conclude that the analysis is sound. This means, for example, that Zero Analysis will never neglect to warn us if we are dividing by a variable that could be zero.

This discussion leads naturally into a fuller treatment of abstract interpretation, which we will turn to in subsequent lectures/readings.